FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Parallelization of the AVL FIRE Benchmark with SVM-Fortran

Michael Gerndt

KFA-ZAM-IB-9520

September 1995
(Stand 13.09.95)

Parallelization of the AVL FIRE Benchmark with SVM-Fortran

Michael Gerndt
Central Institute for Applied Mathematics
Research Centre Jülich (KFA)
52425 Jülich, Germany
m.gerndt@kfa-juelich.de

Abstract

This article outlines the parallelization of an irregular grid application with SVM-Fortran. It describes the different optimizations and their effectiveness. The parallelization was much simplified by the performance analysis tool OPAL, a source code based tool for requesting and analyzing runtime performance data. Although shared memory parallelization is easier than distributed memory parallelization, understanding and eliminating the overhead from page faults is impossible without such a tool. It relates the page faults to the arrays and to the location in the source code. An area which is not supported by OPAL but where supporting tools are highly desirable, is the performance degradation due to low utilization of the on-chip cache.

Keywords: distributed memory multiprocessors, shared virtual memory, parallel programming tools, programming environments, parallel applications

1 Introduction

This paper outlines the parallelization of the AVL FIRE benchmark developed at AVL Graz, Austria. The target system is the Intel Paragon. The parallelization was done using the SVM-Fortran programming environment which is based on an implementation of Shared Virtual Memory [Li 86]. Although the parallelization was facilitated by the shared memory programming model of SVM-Fortran, some optimization had to be performed to take into account the physical distribution of the memory.

The FIRE code is a general purpose computational fluid dynamics program package. It was developed specially for computing compressible and incompressible turbulent fluid flows as encountered in engineering environments.

The benchmark consists of the solver of the resulting linear equation system. The computational domain is discretized with a finite volume approach. The matrices which have to be solved are extremely sparse and have a large and strongly varying bandwidth.

The techniques described in this paper do not handle the irregularity of the grid in a special way. Instead, the techniques are useful for tuning arbitrary SVM-Fortran applications. The performance results are given for the large problem domain (DRALL) which consists of 47312 grid nodes (10 MB of shared memory).

2 Programming SVM on the Intel Paragon

The programming environment we are using on the Intel Paragon is based on SVM. The implementation of the global address space via the paging support of the MACH kernel was done at the Intel European Supercomputer Development Center. Details on the **A**dvanced **S**hared **V**irtual **M**emory system (ASVM) can be found in [Zeis 93]. Due to the memory management overhead on the Intel Paragon, the page fault times are quite high. Page faults requiring communication with other processors typically take 1.5 ms up to 2.5 ms.

On top of the ASVM, KFA implemented a language extension of Fortran77, SVM-Fortran. SVM-Fortran supports shared memory parallel programming via directives. In addition to parallel loops and parallel sections, the user can define the work distribution explicitly. For this purpose, we adapted the template concept of HPF and extended it to make it more flexible. For example, templates can be created dynamically and redistributed at runtime depending on runtime information. Templates direct the distribution of loop iterations onto processors. A detailed description of SVM-Fortran can be found in [BeGe 95], an introduction in [GeBe 95].

SVM-Fortran is supported by two performance analysis tools. The **OP**timizer and Locality **A**nalyzer OPAL is a source code based tool which supports collection and analysis of performance data [GKO 95]. The data are presented by annotating the source code via a separate performance column in the main display or in a separate window if detailed data for individual variables are requested by clicking on a parallel code section. The other tool is the **PAR**allel **Vis**ualization Environment PARvis [NaAr 94] which is a trace visualization tool originally developed at KFA for message passing programs. In addition, it supports views onto data provided by SVM systems and is currently adapted to SVM-Fortran traces.

Both tools allow to analyze data generated by the **S**hared virtual memory **A**pplication **M**onitor SAM [Oz 95, GKO 95]. SAM requires to compile the program once for performance analysis. At runtime, it reads a trace request file which specify the required information and performs an instrumentation according to these requests. This feature allows to reduce the tracing overhead and the amount of trace data significantly. The whole performance analysis for SVM-Fortran is implemented in an incremental way. The user specifies trace requests interactively with the help of OPAL. He clicks on a parallel loop and requests the default set of information (start and stop events for the loop, page fault sums, and synchronization time) or requests individual trace events for page faults of a specific array.

The whole environment is used for about half a year for parallelizing applications. This article outlines some lessons learned from working on the AVL FIRE benchmark. All the optimizations neccessary to obtained good performance for this code are useful for all types of applications. Additional optimzations, like those described in [ToAb 94] taking into account the connections among elements of the finite element grid, have to be investigated in the future.

3 Code Structure

The code consists of four phases:

1. reading the data set
2. initializing data structures
3. solving the system of equations
4. writing the computation time of the solver and the megaflop rate

Due to the shared memory programming model, the input/output need not be transformed at all. The initialization step was parallelized in the same way as the solver and thus needs not be discussed in detail. The interesting part of the program is the solver which uses the truncated Krylov subspace method Orthomin. The outer loop of this algorithm is executed 355 times for this input data set. Two typical code sections in the body of this loop are outlined in Figure 1.

The compute loop is the most time consuming part of the solver. It takes 37.8 seconds out of 67.5 seconds, the sequential execution time for the whole solver on the Intel Paragon. This execution time corresponds to 7.3 MFlops resulting from compiling the code with -O4 and -Mvect. All the measurements were done with this optimization level.

4 Initial Parallelization

In a first step, the code was parallelized with the pdo-directive and the use of reduction variables where it was necessary. For example, the compute loop was annotated with the directive `CSVM$ PDO`, which results in the default block-scheduling strategy. The update step required the use of reduction variables.

```
      occ=0
csvm$ pdo(reduction(occ))
      do nc=nintci,nintcf
         occ=occ+adxor1(nc)*direc2(nc)
      enddo
      oc1=occ/cnorm(1)
csvm$ pdo
      do nc=nintci,nintcf
         direc2(nc)=direc2(nc)-oc1*adxor1(nc)
      enddo
```

The reduction is implemented by computing partial sums in each processor, combining these sums by calling the gdsum function, and copying the resulting global sum to the shared variable in the master processor.

The performance numbers for this version are shown in Table 1. The left columns present the measurements for the whole solver and the right columns the performance number for

```
// compute step
do nc=nintci,nintcf
  direc2(nc)=bp(nc)*direc1(nc)
             - bs(nc)*direc1(lcc(1,nc))
             - be(nc)*direc1(lcc(2,nc))
                ...
             - bh(nc)*direc1(lcc(6,nc))
enddo


// update step
if (nor1.eq.1) then
  oc1=0
  occ=0
  do nc=nintci,nintcf
     occ=occ+adxor1(nc)*direc2(nc)
  enddo
  oc1=occ/cnorm(1)
  do nc=nintci,nintcf
     direc2(nc)=direc2(nc)-oc1*adxor1(nc)
  enddo
  do nc=nintci,nintcf
     direc1(nc)=direc1(nc)-oc1*dxor1(nc)
  enddo
endif
```

Figure 1: Code example from AVL FIRE benchmark

the compute loop only. All times are given in seconds. The values for the page faults and the synchronization are mean values across the processors.

procs	time	MFlops	page faults	sync time	time of compute	page faults compute	sync time compute
1	112.5	4.4	–	–	42.0	–	–
2	100.1	4.9	7025	2.6	31.0	1340	0.3
4	78.1	6.3	8747	12.0	20.1	1520	3.4
8	72.9	6.7	9772	14.3	15.6	1775	3.0
16	79.7	6.2	9785	19.0	13.7	1877	3.0

Table 1: Performance of initial parallel version

This initial parallelization shows very poor performance. The maximum performance is obtained with 8 processors and this is still worse than the performance of the sequential code (67.5 seconds). The compute loop behaves pretty good compared to the overall performance. The single node performance is almost a factor two lower than the performance of the sequential code.

The main reason for this performance is the high number of page faults. This results mainly from false-sharing. For example in the loop:

```
csvm$ pdo
      do nc=nintci,nintcf
         direc2(nc)=direc2(nc)-oc1*adxor1(nc)
      enddo
```

a few pages are accessed by two processors. The accesses result in double page faults since an array element is read before it is written. The little computation in each iteration of such a loop leads to page thrashing depending on the timing of the processors. This thrashing does not happen in every execution and thus measuring the performance of the code is very difficult.

In the compute loop we do have almost the same situation, but here we never saw page thrashing since an iteration consists of a lot of operations. A processor executes the last iterations of his block of iterations when the neighboring processor already has executed the first iterations of its block and thus no thrashing occurs.

In the next section we describe an optimization to reduce the number of page faults.

5 Aligning Loop Iterations and Arrays on Pages

There are two possible optimizations. First, the assignment can be handled as a reduction operation although it is not really a reduction. The compiler would generate a private copy of direc2 for each processor and the individual copies would be combined by the master processor. The drawbacks are the high memory requirement and the distribution of pages of direc2 after the loop. Since the master processor performs the assignment to

the shared array, all pages will be located in his memory and will have to be distributed to the other processors when executing the next parallel loop accessing this array.

We used another optimization which is based on a special scheduling strategy. The arrays are aligned at page boundaries and the loop iterations are distributed among the processors such that a processor executes all iterations accessing the same page. We used the template concept and the general block distribution scheme to implement this optimization. The work distribution is determined by the following directives:

```
csvm$ processors:: p(numprocs())
      integer blocks(16)
csvm$ template:: nodes(:)
csvm$ shared,align::direc2

      call xread

csvm$ create:: nodes(nintci:nintcf)
      call comp_dist(nintci,nintcf,1,numproc(),8,blocks)
csvm$ redistribute (general_block(blocks)) onto p:: nodes
```

The template is created dynamically according to the size of the grid. Subroutine `comp_dist` computes the template distribution. Each array element of `blocks` determines the length of the block assigned to the appropriate processor. The subroutine takes into account the shape of the template, the shape of the processor arrangement, and the data type of the array.

The parallel loops are adapted to the distribution via predefined scheduling:

```
csvm$ pdo(loops(nc),strategy(on_home(nodes(nc))))
      do nc=nintci,nintcf
         direc2(nc)=direc2(nc)-oc1*adxor1(nc)
      enddo
```

This optimization results in the performance shown in Table 2. The performance for 8 and 16 processors is better than in the previous version since the number of page faults in each processor was significantly reduced. The execution time of the single node version and on smaller processor numbers is worse since the alignment of arrays at page boundaries reduces the utilization of the on-chip cache.

6 Reduction of Synchronization, Optimizing Control Flow, Privatizing Scalars

There are several other overheads limiting the speedup. The current version contains a lot of barrier operations due to the semantics of the parallel loops. Almost all barriers can be eliminated since no dependences exist across processors except reductions. Only one barrier, prior to the compute step, has to remain. In addition, the emulation of the control

procs	time	MFlops	page faults	sync time	time of compute	page faults compute	sync time compute
1	197.8	2.5	–	–	121.4	–	–
2	119.8	4.1	3320	5.9	69.4	1002	3.3
4	72.5	6.8	3135	7.8	38.8	1014	3.8
8	49.8	9.9	3198	7.9	22.8	1100	2.8
16	45.1	10.9	2992	9.8	15.3	1139	2.4

Table 2: Performance of aligned parallel version

flow in exclusive mode is expensive. By transforming the whole solver into a replicated region the flow of control is determined in each processor independently.

```
csvm$ replicated_region(private(oc1,occ,help))
       ...
csvm$ barrier
csvm$ pdo(loops(nc),strategy(on_home(nodes(nc))),nobarrier)
      do nc=nintci,nintcf
         direc2(nc)=bp(nc)*direc1(nc)
                    - bs(nc)*direc1(lcc(1,nc))
                      ...
      enddo


      // update step
      if (nor1.eq.1) then
        oc1=0
        occ=0
csvm$   pdo(loops(nc),strategy(on_home(nodes(nc)))
csvm$*      reduction(occ),nobarrier)
        do nc=nintci,nintcf
           occ=occ+adxor1(nc)*direc2(nc)
        enddo
        oc1=occ/cnorm(1)
        ...
      endif

csvm$ replicated_region_end
```

In addition, scalars were privatized such that page faults due to the implementation of reductions were eliminated.

The performance numbers shown in Table 3 show the much better speedup due to these optimizations.

procs	time	MFlops	page faults	sync time	time of compute	page faults compute	sync time compute
1	185.4	2.7	–	–	122.4	–	–
2	107.6	4.6	2631	1.3	67.7	979	–
4	59.3	8.3	2353	0.8	34.5	1003	–
8	37.1	13.2	2362	1.0	20.0	1095	–
16	24.9	19.7	2129	0.6	12.9	1137	–

Table 3: Performance after single node optimization

7 Optimizing Cache Performance

Most of the execution time is spent in the compute loop. Due to the cache performance, the speedup of this loop is not optimal. We optimized the cache behaviour by reducing the alignment. The arrays storing the coefficients and the index patterns are only read and thus are not critical to false-sharing. The alignment of these arrays was eliminated resulting in the performance shown in Table 4.

procs	time	MFlops	page faults	sync time	time of compute	page faults compute	sync time compute
1	104.3	4.7	–	–	43.0	–	–
2	63.2	7.8	2702	0.5	26.7	979	–
4	38.1	12.9	2362	0.7	14.7	1006	–
8	25.8	19.0	2383	0.6	9.9	1100	–
16	18.3	26.8	2147	0.7	7.8	1142	–

Table 4: Performance after cache optimization

8 Prefetching and Monitoring Overhead

In the compute loop, the processors read few pages of the neighbouring processors. Therefore, write permission to these pages is reduced to read-only in the other processors and they have to upgrade the permission again when executing subsequent loops. To reduce the page fault time for these accesses, we tried to exploit the new implementation of prefetching.

We first tried the following block prefetch:

```
csvm$ pdo(loops(nc),strategy(on_home(nodes(nc))),nobarrier)
      do nc=nintci,nintcf
         direc2(nc)=bp(nc)*direc1(nc)
                    - bs(nc)*direc1(lcc(1,nc))
                      ...
      enddo
      call prefetch(direc1(first),direc1(last),WRITE)
```

`First` and `last` are the indices of the first written array element and the last written array element. The pages with read-only permission are prefetched with write permission. Due to the overhead of the block-prefetch, the execution time on 16 processors was increased from 18.3 seconds to 22.5 seconds.

We also tried a version where only those pages are prefetched which are on the boundaries of the accessed array section. This implementation is faster than the block prefetch routine since the state of the intermediate pages need not be checked.

```
csvm$ pdo(loops(nc),strategy(on_home(nodes(nc))),nobarrier)
      do nc=nintci,nintcf
         direc2(nc)=bp(nc)*direc1(nc)
                    - bs(nc)*direc1(lcc(1,nc))
                      ...
      enddo
      call prefetch(direc1(first),
                    direc1(first+1024),
                    direc1(last-1024),
                    direc1(last),WRITE,1)
```

The execution time of this version was 19.8 seconds and thus similar to the version without prefetching (18.3 seconds). In general, the reason for the worse performance with prefetching is the additional overhead and the lack of computation between the prefetch and the code accessing the missing pages.

We also tested the monitoring overhead since all the measurements were done with monitoring support to obtain the performance number for the whole program and the compute loop. The execution time of the program without monitoring on 16 processors was 18.2 seconds and on 8 processors 24.8 seconds. The overhead due to the monitoring is near to the accuracy of the measurements and thus not significant.

9 Conclusions

All the optimizations performed during the parallelization are relevant to all types of applications. The performance of the AVL FIRE benchmark after the different optimizations is summarized in Table 4. Up to now, special optimizations for handling the sparsity of the matrix and thus, to reduce the number of page faults in the compute loop, have not been performed. It took quite some effort to apply the standard optimizations.

procs	initial version	aligned version	synchronization, privatization	cache opt., final version
1	4.4	2.5	2.7	4.7
2	4.9	4.1	4.6	7.8
4	6.3	6.8	8.3	12.9
8	6.7	9.9	13.2	19.0
16	6.2	10.9	19.7	26.8

Table 5: Performance after different optimizations (MFlops)

The performance analysis tool OPAL was absolutly necessary in the parallelization process. It was only possible to understand the performance bottlenecks of the code by using this tool showing the page faults and the synchronization overhead. During the parallelization, the tool was extended in three ways:

1. Analyzing the synchronization time

   The standard instrumentation request now includes the time for synchronizations. The information can be inspected via the summary, loop analysis dialog, and the information column.

2. Translation of page faults to array elements

   A request for individual page faults now consists also of the variable mapping request. During the inspection of page faults in the loop analysis dialog, the mapping information for the loaded procedure is updated via the variable mapping events. The faulting address is searched in the symbol table. If a variable is found and the symbol is a fixed size array, the name and the indices are shown.

3. Overhead computation

   When analyzing the overhead of the parallel execution, the individual sums of the page fault time and the synchronization time are misleading. The overhead is the sum of both values in each processor which may be different from the sum of the mean values of both times across all processes. Therefore, a new analysis was integrated that computes the overhead (sum of page fault time and synchronization time) in each processor. The maximum overhead can be inspected via the performance column or the individual overhead in each processor via the popup menu.

After identifying page faults for an array such as direc1, it is important to understand where the page faults occur and where the pages or the permissions are lost. This is

supported quite well in OPAL since all paging information for an individual array can be requested. When analyzing the performance information, the page faults and the invalidations or permission reductions for the pages can be inspected and the corresponding processor identified. If the information is generated for multiple arrays, the inspection via the loop analysis dialog can be restricted to individual arrays.

A problem with using OPAL is to remember the instrumentation. This problem gets worse when multiple trace files are available. Some support has to be integrated into OPAL to show the requests and to restrict the analysis features to those applicable to the requested information. For example, the shared variable list in the loop analysis dialog should be restricted to those variables for which the performance information was generated.

Similar to the parallelization and optimization of the crystal growth program[GeBe 95], the performance is influenced very much by the cache performance which is reduced by aligning arrays to eliminate false-sharing. Appropriate support in OPAL for detecting such situations would be extremely useful. Currently, the only hint for such an effect is an increased execution time together with a decreased overhead when applying the described optimization.

The remaining page faults due to accesses in the compute loop were difficult to understand. The structure of the code is a little complicated since the control flow is different in each iteration of the outer loop. In one iteration the write upgrades happen in a code section with an update step, in the next iteration they happen in an update step in a different code section, and in the third iteration they happen in the loop before the compute loop. This complex cyclical situation can only be understood when analyzing the dynamic behaviour of the code and thus will best be analyzed by using PARvis. The analysis would be facilitated by an abstract visualization of the control flow.

References

[BeGe 95] R. Berrendorf, M. Gerndt, *SVM-Fortran Reference Manual Version 1.4*, Internal Report KFA-ZAM-IB-9510, Central Institute for Applied Mathematics, Research Centre Jülich, 1995

[GKO 95] M. Gerndt, A. Krumme, S. Özmen, *Performance Analysis for SVM-Fortran with OPAL*, Submitted for publication to International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Georgia, 1995

[GeBe 95] M. Gerndt, R. Berrendorf , *Parallelizing Applications with SVM-Fortran*, Proceedings of the HPCN'95, Mailand, LNCS 919, pp. 793 - 798, 1995

[Li 86] Kai Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Dissertation, Yale University 1986, Technical Report YALEU/DCS/RR-492

[NaAr 94] W.E. Nagel, A. Arnold, *Performance Visualization of Parallel Programs - The PARvis Environment* -, Proceedings 1994 Intel Supercomputing Users Group (ISUG), pp. 24-31, 1994

[Oz 95] S. Özmen, *SAM: Performance-Analyse-Monitor für SVM-Fortran*, Diploma Thesis, RWTH Aachen, 1995

[ToAb 94] K.A. Tomko, S.G. Abraham, *Data and Program Restructuring of Irregular Applications for Cache-Coherent Multiprocessors*, Proceedings ICS94, Manchester, pp. 214-225, 1994

[Zeis 93] S. Zeisset, *Evaluation and Enhancement of the Paragon Multiprocessor's Shared Virtual Memory System*, Diploma Thesis, Technische Universität München, 1993